LinuxFocus article number 266
http://linuxfocus.org

Developing Applications for Gnome with Python (Part 3)

by Hilaire Fernandes
<hilaire(at)ofset.org>

About the author:

Hilaire Fernandes is the Vice-President of OFSET, an organization to promote the development of 'Free' educational software for the Gnome desktop. He also wrote Dr. Geo, a primer program for dynamic geometry, and is currently working on Dr. Genius - another education program for Gnome.

Abstract:

This series of articles is specially written for newbie programmers using Gnome and GNU/Linux. Python, the chosen language for development, avoids the usual overhead of compiled languages like C. To understand this article you need a basic understanding of Python programming. More information on Python and Gnome are available at http://www.python.org and http://www.gnome.org.

Previous articles in the series :
- first article
- second article

Translated to English by:
Lorne Bailey
<sherm_pbody(at)yahoo.com>

Required tools

For the software dependencies needed to execute the program described in this article, please refer to the list from part I of this series of articles.

You will also need:

- In the original .glade file[drill.glade] . This file has been slightly modified since last time to incorporate sliders to choose exercises in the interface.

- This time the Python source code is distributed in four files :
    1. [drill1.py].
    2. [templateExercice.py].
    3. [colorExercice.py].
    4. [labelExercice.py].

For installation and use of Python-Gnome and LibGlade please refer to Part I.

Development Model for the Exercises

In the preceeding article (part 2), we created the user interface -- **Drill** -- which is a frame for the coding of the exercises described further on. Now, we shall take a closer look at object oriented development using Python, in order to add functionalities to **Drill**. In this study, we will leave aside the aspects of Python development in Gnome.

So let's pick up where we left off, the insertion of a color game into **Drill** as an exercise for the reader. We will use this to illustrate our current subject and at the same time offer a solution for that exercise.

Object Oriented Development

Briefly, without claiming to make an exhaustive analysis, object oriented development attempts to define and categorize things by **is a** relationships, whether they exist in the physical world or not. This can be seen as abstracting the objects related to the problem in which we're interested. We can find comparisons in different domains like the categories of Aristotle, taxonomies, or ontologies. In each case, one must understand a complex situation through an abstraction. This type of development could very well have been called category oriented development.

In this development model, objects manipulated by the program, or constituting the program, are called **classes** and representatives of these abstract objects are **instances**. Classes are defined by **attributes** (containing values) and **methods** (functions). We speak of a parent-child relationship for a given class where a child class inherits properties from a parent. Classes are organized by an **is a** relationship, where a child still **is a** type of the parent as well as a child type. Classes might not be completely defined, in which case they are called abstract classes. When a method is declared but not defined (the body of the function is void) it is also called a virtual method. An abstract class has one or more of these undefined methods and therefore cannot be instantiated. Abstract classes permit specification of the form taken by derived classes - child classes in which the pure virtual methods will be defined.

Different languages have more or less elegance in defining objects, but the common denominator seems to be the following :

1. Inheritance of attributes and methods of the parent class by the child.
2. Ability of the child class to override and overload the methods inherited from the parent.
3. Polymorphism, where a class might have many parent classes.

Python and Object Oriented Development

In the case of Python, this is the lowest common denominator that has been chosen. This permits learning object oriented development without getting lost in the details of this methodology.

In Python, an object's methods are always virtual methods. This means they can always be overridden by a child class -- which is generally what we want using object oriented development -- and which slightly simplifies the syntax. But it's not easy to distinguish between methods that are overridden or not. Furthermore it is impossible to render an object opaque and therefore deny access to attributes and methods from outside an object. In conclusion, attributes of a Python object are both readable and writable from outside the object.

Parent Class exercice

In our example, (see the file `templateExercice.py`), we would like to define many objects of the type `exercice`. We define an object of type `exercice` to serve as an abstract base class for deriving other exercises that we will create later. The object `exemple` is the parent class of all the other types of exercises created. These derived types of exercises will have at least the same attributes and methods as the class `exercice` because they will inherit them. This will permit us to manipulate all the diverse types of exercice objects identically, regardless of the object they are instantiated from.

For example, to create an instance of the class `exercice` we can write :

```
from templateExercice import exercice

monExercice = exercice ()
monExercice.activate (ceWidget)
```

In fact, there's no need to create an instance of the class `exercice` because it's only a template from which other classes are derived.

Attributes

- `exerciceWidget` : the widget containing the exercise's user interface ;
- `exerciceName` : the name of the exercise.

If we are interested in other aspects of an exercise we can add attributes, e.g. the score obtained or the number of times it has been run.

Methods

- `__init__ (self)`: this method has a very precise role in a Python object. It is automatically called during the creation of an instance of this object. For this reason it is also called the constructor. The argument `self` is a reference to the instance of the class `exercice` that called the `__init__` method. It is always necessary to specify this argument in methods, which means that a method cannot have zero arguments. Careful, this argument is added automatically by Python, so it is not necessary to include it when calling the method. The argument `self` allows access to the attributes and other methods of an instance. Without it, such access is impossible. We will see that in greater detail later.
- `activate (self, area)` : activates this instance of exercice by placing its widget in the exercice zone of **Drill**. The argument `area` is actually a GTK+ container that controls the widget's placement in **Drill**. Knowing that the attribute `exerciceWidget` contains the exercise's widget, one need only call `area.add (self.exerciceWidget)` to wrap the exercice in **Drill**.
- `unactivate (self, area)` : removes the widget from the container **Drill**. In terms of placement, this is the opposite operation, so calling `area.remove (self.exerciceWidget)` will suffice.
- `reset (self)` : reset the exercise to zero.

In Python code this gives you :

```
class exercice:
    "A template exercice"
    exerciceWidget = None
    exerciceName = "No Name"
    def __init__ (self):
        "Create the exericice widget"
    def activate (self, area):
        "Set the exercice on the area container"
        area.add (self.exerciceWidget)
    def unactivate (self, area):
        "Remove the exercice fromt the container"
        area.remove (self.exerciceWidget)
    def reset (self):
        "Reset the exercice"
```

This code is included in its own file `templateFichier.py`, which permits us to clarify the specific roles of each object. The methods are declared inside the class `exercice`, and are in fact functions.

We will see that the argument `area` is a reference to a GTK+ widget constructed by LibGlade, it's a window with sliders.

In this object, the methods `__init__` and `reset` are empty and will be overridden by the child classes if necessary.

labelExercice, First Example of Inheritance

This is almost an empty exercise. It only does one thing, it puts the name of the exercise into the

exercise zone of **Drill** . It serves as a starter for the exercises that populate the left-hand tree of **Drill** but that we haven't created yet.

In the same way as the object `exercice`, the object `labelExercice` is put in it's own file, `labelExercice.py` . Next, since this object is a child of the object `exercice` , we need to tell it how the parent is defined. This is done simply by an import :

```
from templateExercice import exercice
```

This literally means that the definition of the class `exercice` in the file `templateExercice.py` is imported in the current code.

We come now to the most important aspect, the declaration of the class `labelExercice` as a child class of `exercice`.
`labelExercice` is declared in the following fashion :

```
class labelExercice(exercice):
```

Voilà, that's enough so that `labelExercice` inherits all the attributes and methods of `exercice`.

Of course we still have work to do, in particular we need to initialize the widget of the exercise. We do this by overriding the method `__init__` (i.e. in redefining it in the class `labelExercice`), this last is called when an instance is created. Also, this widget must be referenced in the attribute `exerciceWidget` so we will not need to override the `activate` and `unactivate` methods of the parent class `exercice`.

```
  def __init__ (self, name):
      self.exerciceName = "Un exercice vide" (Trans. note: an empty exercise)
      self.exerciceWidget = GtkLabel (name)
      self.exerciceWidget.show ()
```

This is the only method that we override. To create an instance of `labelExercice`, one need only call :

```
monExercice = labelExercice ("Un exercice qui ne fait rien")
(Translator Note: "Un exercice qui ne fait rien" means "an exercise doing nothing")
```

To access it's attributes or methods :

```
# Le nom de l'exercice (Translator Note: name of the exercise)
print monExercice.exerciceName

# Placer le widget de l'exercice dans le container "area"
# (Translator Note: place the exercise's
widget in the container "area")
monExerice.activate (area)
```

colorExercice, Second Example of Inheritance

Here we begin the transformation of the color game seen in the first article of this series into a class of type `exercice` that we will name `colorExercice`. We place it in it's own file, `colorExercice.py` , that is appended to this article with complete source code.

The changes required to the initial source code consist mostly of a redistribution of functions and variables into methods and attributes in the class `colorExercice`.

The global variables are transformed into attributes declared at the beginning of the class :

```
class colorExercice(exercice):
    width, itemToSelect = 200, 8
    selectedItem = rootGroup = None
    # to keep trace of the canvas item
    colorShape = []
```

Like for the class `labelExercice`, the method `__init__` is overridden to accommodate the construction of the exercise's widgets :

```
def __init__ (self):
    self.exerciceName = "Le jeu de couleur" # Translator Note: the color game
    self.exerciceWidget = GnomeCanvas ()
    self.rootGroup = self.exerciceWidget.root ()
    self.buildGameArea ()
    self.exerciceWidget.set_usize (self.width,self.width)
    self.exerciceWidget.set_scroll_region (0, 0, self.width, self.width)
    self.exerciceWidget.show ()
```

Nothing new compared to the initial code if it's only the `GnomeCanvas` referenced in the attribute `exerciceWidget`.

The other overridden method is `reset`. Since it resets the game to zero, it must be customized for the color game :

```
    def reset (self):
        for item in self.colorShape:
            item.destroy ()
        del self.colorShape[0:]
        self.buildGameArea ()
```

The other methods are direct copies of the original functions, with the added use of the variable `self` to allow access to the attributes and methods of the instance. There is one exception in the methods `buildStar` and `buildShape` where the decimal parameter `k` is replaced by a whole number. I noted strange behavior in the document `colorExercice.py` where the decimal numbers grabbed by the source code are truncated. The problem seems to be tied to the module `gnome.ui` and to the French locale (where decimal numbers use a comma for a separator instead of a period). I will work at finding the

source of the problem before the next article.

Final adjustments in Drill

We now have two types of exercise -- `labelExercice` and `colorExercice`. We create instances of them with the functions `addXXXXExercice` in the code `drill1.py`. The instances are referenced in a dictionary `exerciceList` in which the keys are also arguments to the pages of each exercise in the tree at left:

```
def addExercice (category, title, id):
    item = GtkTreeItem (title)
    item.set_data ("id", id)
    category.append (item)
    item.show ()
    item.connect ("select", selectTreeItem)
    item.connect ("deselect", deselectTreeItem)
[...]
def addGameExercice ():
    global exerciceList
    subtree = addSubtree ("Jeux")
    addExercice (subtree, "Couleur", "Games/Color")
    exerciceList ["Games/Color"] = colorExercice ()
```

The function `addGameExercice` creates a leaf in the tree with the attribute `id="Games/Color"` by calling the function `addExercice`. This attribute is used as a key for the instance of the color game created by the command `colorExercice()` in the dictionary `exerciceList`.

Next, due to the elegance of polymorphism in object oriented development, we can run the exercises by using same functions that act differently for each object without worrying about their internal implementation. We only call methods defined in the abstract base class `exercice` and they do different things in class colorExercice or labelExercice. The programmer "speaks" to all the exercises in the same way, even if the "response" of each exercise is a little different. To do this we combine the use of the attribute `id` of the pages of the tree and the dictionary `exerciceList` or the variable `exoSelected` that refers to the exercise in use. Given that all the exercises are children of the class `exercice`, we use its methods the same way to control the exercises in all their variety.

```
def on_new_activate (obj):
    global exoSelected
    if exoSelected != None:
        exoSelected.reset ()

def selectTreeItem (item):
    global exoArea, exoSelected, exerciceList
    exoSelected = exerciceList [item.get_data ("id")]
    exoSelected.activate (exoArea)

def deselectTreeItem (item):
    global exoArea, exerciceList
    exerciceList [item.get_data ("id")].unactivate (exoArea)
```

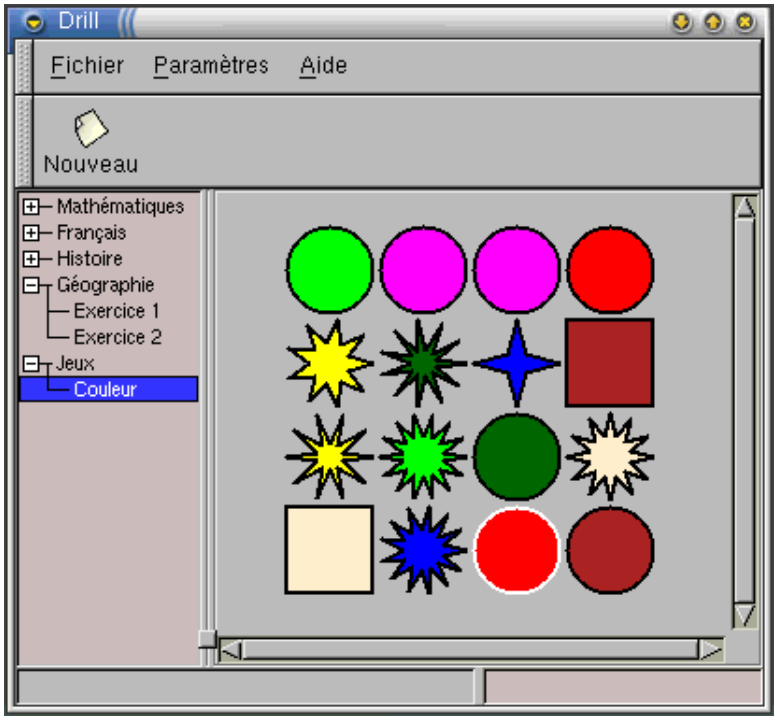

Fig. 1 - Main window of Drill, with the color exercise

Thus ends our article. We have discovered the attractions of object oriented development in Python within the realm of a graphical user interface. In the next articles, we will continue discovering Gnome widgets through coding new exercises that we will insert into **Drill**.

Appendix: Complete Source Code

drill1.py

```
#!/usr/bin/python
# Drill - Teo Serie
# Copyright Hilaire Fernandes 2002
# Release under the terms of the GPL licence
# You can get a copy of the license at http://www.gnu.org


from gnome.ui import *
from libglade import *

# Import the exercice class
from colorExercice import *
from labelExercice import *

exerciceTree = currentExercice = None
# The exercice holder
exoArea = None
exoSelected = None
```

```
exerciceList = {}

def on_about_activate(obj):
    "display the about dialog"
    about = GladeXML ("drill.glade", "about").get_widget ("about")
    about.show ()

def on_new_activate (obj):
    global exoSelected
    if exoSelected != None:
        exoSelected.reset ()

def selectTreeItem (item):
    global exoArea, exoSelected, exerciceList
    exoSelected = exerciceList [item.get_data ("id")]
    exoSelected.activate (exoArea)

def deselectTreeItem (item):
    global exoArea, exerciceList
    exerciceList [item.get_data ("id")].unactivate (exoArea)

def addSubtree (name):
    global exerciceTree
    subTree = GtkTree ()
    item = GtkTreeItem (name)
    exerciceTree.append (item)
    item.set_subtree (subTree)
    item.show ()
    return subTree

def addExercice (category, title, id):
    item = GtkTreeItem (title)
    item.set_data ("id", id)
    category.append (item)
    item.show ()
    item.connect ("select", selectTreeItem)
    item.connect ("deselect", deselectTreeItem)


def addMathExercice ():
    global exerciceList
    subtree = addSubtree ("Mathématiques")
    addExercice (subtree, "Exercice 1", "Math/Ex1")
    exerciceList ["Math/Ex1"] = labelExercice ("Exercice 1")
    addExercice (subtree, "Exercice 2", "Math. Ex2")
    exerciceList ["Math/Ex2"] = labelExercice ("Exercice 2")

def addFrenchExercice ():
    global exerciceList
    subtree = addSubtree ("Français")
    addExercice (subtree, "Exercice 1", "French/Ex1")
    exerciceList ["French/Ex1"] = labelExercice ("Exercice 1")
    addExercice (subtree, "Exercice 2", "French/Ex2")
    exerciceList ["French/Ex2"] = labelExercice ("Exercice 2")

def addHistoryExercice ():
    global exerciceList
    subtree = addSubtree ("Histoire")
    addExercice (subtree, "Exercice 1", "Histoiry/Ex1")
    exerciceList ["History/Ex1"] = labelExercice ("Exercice 1")
    addExercice (subtree, "Exercice 2", "Histoiry/Ex2")
```

```
    exerciceList ["History/Ex2"] = labelExercice ("Exercice 2")

def addGeographyExercice ():
    global exerciceList
    subtree = addSubtree ("Géographie")
    addExercice (subtree, "Exercice 1", "Geography/Ex1")
    exerciceList ["Geography/Ex1"] = labelExercice ("Exercice 1")
    addExercice (subtree, "Exercice 2", "Geography/Ex2")
    exerciceList ["Geography/Ex2"] = labelExercice ("Exercice 2")

def addGameExercice ():
    global exerciceList
    subtree = addSubtree ("Jeux")
    addExercice (subtree, "Couleur", "Games/Color")
    exerciceList ["Games/Color"] = colorExercice ()


def initDrill ():
    global exerciceTree, label, exoArea
    wTree = GladeXML ("drill.glade", "drillApp")
    dic = {"on_about_activate": on_about_activate,
           "on_exit_activate": mainquit,
           "on_new_activate": on_new_activate}
    wTree.signal_autoconnect (dic)
    exerciceTree = wTree.get_widget ("exerciceTree")
    # Temporary until we implement real exercice
    exoArea = wTree.get_widget ("exoArea")
    # Free the GladeXML tree
    wTree.destroy ()
    # Add the exercice
    addMathExercice ()
    addFrenchExercice ()
    addHistoryExercice ()
    addGeographyExercice ()
    addGameExercice ()

initDrill ()
mainloop ()
```

templateExercice.py

```
# Exercice pure virtual class
# exercice class methods should be override
# when exercice class is derived
class exercice:
    "A template exercice"
    exerciceWidget = None
    exerciceName = "No Name"
    def __init__ (self):
        "Create the exericice widget"
    def activate (self, area):
        "Set the exercice on the area container"
        area.add (self.exerciceWidget)
    def unactivate (self, area):
        "Remove the exercice fromt the container"
        area.remove (self.exerciceWidget)
    def reset (self):
        "Reset the exercice"
```

labelExercice.py

```
# Dummy Exercice - Teo Serie
# Copyright Hilaire Fernandes 2001
# Release under the terms of the GPL licence
# You can get a copy of the license at http://www.gnu.org

from gtk import *
from templateExercice import exercice

class labelExercice(exercice):
    "A dummy exercie, it just prints a label in the exercice area"
    def __init__ (self, name):
        self.exerciceName = "Un exercice vide"
        self.exerciceWidget = GtkLabel (name)
        self.exerciceWidget.show ()
```

colorExercice.py

```
# Color Exercice - Teo Serie
# Copyright Hilaire Fernandes 2001
# Release under the terms of the GPL licence
# You can get a copy of the license at http://www.gnu.org

from math import cos, sin, pi
from whrandom import randint
from GDK import *
from gnome.ui import *

from templateExercice import exercice


# Exercice 1 : color game

class colorExercice(exercice):
    width, itemToSelect = 200, 8
    selectedItem = rootGroup = None
    # to keep trace of the canvas item
    colorShape = []
    def __init__ (self):
        self.exerciceName = "Le jeu de couleur"
        self.exerciceWidget = GnomeCanvas ()
        self.rootGroup = self.exerciceWidget.root ()
        self.buildGameArea ()
        self.exerciceWidget.set_usize (self.width,self.width)
        self.exerciceWidget.set_scroll_region (0, 0, self.width, self.width)
        self.exerciceWidget.show ()
    def reset (self):
        for item in self.colorShape:
            item.destroy ()
        del self.colorShape[0:]
        self.buildGameArea ()
    def shapeEvent (self, item, event):
```

```
    if event.type == ENTER_NOTIFY and self.selectedItem != item:
        item.set(outline_color = 'white') #highligh outline
    elif event.type == LEAVE_NOTIFY and self.selectedItem != item:
        item.set(outline_color = 'black') #unlight outline
    elif event.type == BUTTON_PRESS:
        if not self.selectedItem:
            item.set (outline_color = 'white')
            self.selectedItem = item
        elif item['fill_color_gdk'] == self.selectedItem['fill_color_gdk'] \
             and item != self.selectedItem:
            item.destroy ()
            self.selectedItem.destroy ()
            self.colorShape.remove (item)
            self.colorShape.remove (self.selectedItem)
            self.selectedItem, self.itemToSelect = None, \
             self.itemToSelect - 1
            if self.itemToSelect == 0:
                self.buildGameArea ()
    return 1

def buildShape (self,group, number, type, color):
    "build a shape of 'type' and 'color'"
    w = self.width / 4
    x, y, r = (number % 4) * w + w / 2, (number / 4) * w + w / 2, w / 2 - 2
    if type == 'circle':
        item = self.buildCircle (group, x, y, r, color)
    elif type == 'squarre':
        item = self.buildSquare (group, x, y, r, color)
    elif type == 'star':
        item = self.buildStar (group, x, y, r, 2, randint (3, 15), color)
    elif type == 'star2':
        item = self.buildStar (group, x, y, r, 3, randint (3, 15), color)
    item.connect ('event', self.shapeEvent)
    self.colorShape.append (item)

def buildCircle (self,group, x, y, r, color):
    item = group.add ("ellipse", x1 = x - r, y1 = y - r,
                      x2 = x + r, y2 = y + r, fill_color = color,
                      outline_color = "black", width_units = 2.5)
    return item

def buildSquare (self,group, x, y, a, color):
    item = group.add ("rect", x1 = x - a, y1 = y - a,
                      x2 = x + a, y2 = y + a, fill_color = color,
                      outline_color = "black", width_units = 2.5)
    return item

def buildStar (self,group, x, y, r, k, n, color):
    "k: factor to get the internal radius"
    "n: number of branch"
    angleCenter = 2 * pi / n
    pts = []
    for i in range (n):
        pts.append (x + r * cos (i * angleCenter))
        pts.append (y + r * sin (i * angleCenter))
        pts.append (x + r / k * cos (i * angleCenter + angleCenter / 2))
        pts.append (y + r / k * sin (i * angleCenter + angleCenter / 2))
    pts.append (pts[0])
    pts.append (pts[1])
    item = group.add ("polygon", points = pts, fill_color = color,
                      outline_color = "black", width_units = 2.5)
```

```
        return item

    def getEmptyCell (self,l, n):
        "get the n-th non null element of l"
        length, i = len (l), 0
        while i < length:
            if l[i] == 0:
                n = n - 1
            if n < 0:
                return i
            i = i + 1
        return i

    def buildGameArea (self):
        itemColor = ['red', 'yellow', 'green', 'brown', 'blue', 'magenta',
                     'darkgreen', 'bisque1']
        itemShape = ['circle', 'squarre', 'star', 'star2']
        emptyCell = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
        self.itemToSelect, i, self.selectedItem = 8, 15, None
        for color in itemColor:
            # two items of same color
            n = 2
            while n > 0:
                cellRandom = randint (0, i)
                cellNumber = self.getEmptyCell (emptyCell, cellRandom)
                emptyCell[cellNumber] = 1
                self.buildShape (self.rootGroup, cellNumber, \
                 itemShape[randint (0, 3)], color)
                i, n = i - 1, n - 1
```


2005-01-14, generated by lfparser_pdf version 2.51